Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair

Anonymous Author(s)

ABSTRACT

A large body of the literature of automated program repair develops approaches where patches are generated to be validated against an oracle (e.g., a test suite). Because such an oracle can be imperfect, the generated patches, although validated by the oracle, may actually be incorrect. While the state of the art explore research directions that require dynamic information or that rely on manually-crafted heuristics, we study the benefit of learning code representations in order to learn deep features that may encode the properties of patch correctness. Our empirical work mainly investigates different representation learning approaches for code changes to derive embeddings that are amenable to similarity computations. We report on findings based on embeddings produced by pre-trained and re-trained neural networks. Experimental results demonstrate the potential of embeddings to empower learning algorithms in reasoning about patch correctness: a machine learning predictor with BERT transformer-based embeddings associated with logistic regression yielded an AUC value of about 0.8 in the prediction of patch correctness on a deduplicated dataset of 1000 labeled patches. Our investigations show that learned representations can lead to reasonable performance when comparing against the state-of-the-art, PATCH-SIM, which relies on dynamic information. These representations may further be complementary to features that were carefully (manually) engineered in the literature.

1 INTRODUCTION

Automation in software engineering has recently reached new heights with the promising results recorded in the research direction of program repair [25, 38]. While a few techniques try to model program semantics and synthesize execution constraints towards producing quality patches, they often fail to scale to large programs. Instead, the large majority of research contributions [37] explore search-based approaches where patch candidates are generated and then validated against an oracle.

In the absence of strong program specifications, test suites represent affordable approximations that are widely used as the oracle in program repair. In their seminal approach to test-based program repair, Weimer *et al.* [49] considered that a patch is acceptable as soon as it makes the program pass all test cases in the test suite. Since then, a number of studies [41, 44] have explored the *overfitting* problem in patch validation: a given patch is synthesized to pass a test suite and yet is incorrect with respect to the intended program specification. Since limited test suites only weakly approximate program specifications, a patched program can indeed satisfy the requirements encoded in the test cases, and present a behavior outside of those tests that are significantly different from the behavior initially expected by the developer.

ASE 2020, September 21–25, 2020, Melbourne, Australia
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Overfitting patches constitute a key challenge in generate-and-validate approaches of automated program repair (APR). Recent evaluation campaigns [16, 29, 30, 43, 50] on APR systems are stressing on the importance of estimating the correctness ratio among the valid patches that can be found. To improve this ratio, researchers are investigating several research directions. We categorize them in three main axes that focus on actions before, during or after patch generation:

- *test-suite augmentation:* Yang *et al.* [56] proposed to generate better test cases to enhance the validation of patches, while Xin and Reiss [52] opted for increasing test inputs.
- *curation of repair operators:* approaches such as CapGen [50] successfully demonstrated that carefully-designed (e.g., fine-grained fix ingredients) repair operators can lead to more correct patches.
- *post-processing of generated patches:* Long and Rinard [32] have explored some heuristics to discard patches that are likely overfitting.

Our work is related to the latter thrust. So far, the state-of-the-art works targeting the identification of patch correctness are mainly implemented based on computing the similarity of test case execution traces [53]. Ye *et al.* [57] followed up by presenting preliminary results suggesting that statically-extracted code features at the syntax level could be used to predict overfitting patches. While such an approach is appealing, the feature engineering effort can be huge when researchers target generalizable approaches. To cope with this problem, Csuvik *et al.* [8] have proposed a preliminary small-scale study on the use of embeddings: leveraging pre-trained natural language sentence embedding models, they claim to have been able to filter out 45% incorrect patches generated for 40 bugs from the QuixBugs dataset [58].

This paper. Embeddings have been successfully applied to various prediction tasks in software engineering research [1, 27, 45, 46]. For patch correctness prediction, the literature does not yet provide extensive experimental results to guide future research. Our work fills this gap. We investigate in this paper the feasibility of leveraging advances in deep representation learning to produce embeddings that are amenable to reasoning about correctness.

❶ We investigate different representation learning models adapted to natural language tokens and source code tokens that are more specialized to code changes. Our study considers both pre-trained models and the retraining of models.
❷ We empirically investigate whether, with learned representations, the hypothesis of minimal changes incurred by correct patches remains valid: experiments are performed to check the statistical difference between similarity scores yielded by correct patches and those yielded by incorrect patches.
❸ We run exploratory experiments assessing the possibility to select cutoff similarity scores between learned representations of buggy code and patched code fragments for heuristically filtering out incorrect patches.

❹ Finally, we investigate the discriminative power of deep learned features in a classification training pipeline aimed at learning to predict patch correctness.

2 BACKGROUND

Our work deals with various concepts and techniques from the fields of program repair and machine learning. We present the relevant details in this section to facilitate readers' understanding of our study design and the scope of our experiments.

2.1 Patch Plausibility and Correctness

Defining patch correctness is a non-trivial challenge in automated program repair. Until the release of empirical investigations by Smith *et al.* [44], actual correctness (w.r.t. program behavior intended by developers) was seldom used as a performance criterion of patch generation systems. Instead, experimental results were focused on the number of patches that make the program pass all test cases. Such patches are actually only **plausible**. Qi *et al.* [41] demonstrated in their study that an overwhelming majority of plausible patches generated by GenProg [24], RSRepair [40] and AE [48]) are overfitting the test suite while actually being incorrect.

To improve the probability of program repair systems to generate **correct** patches, researchers have mainly invested in strengthening the validation oracle (i.e., the test suites). Opad [56], DiffTGen [52], UnsatGuided [60], PATCH-SIM/TEST-SIM [53] generate new test inputs that trigger behavior cases which are not addressed by APR-generated patches.

More recent works [8, 57] are starting to investigate static features and heuristics (or machine learning) to build predictive models of patch correctness. Ye *et al.* [57] presented the ODS approach which relates to our study since it investigated machine learning with static features extracted from Java program patches. Their approach however builds on carefully hand-crafted features, which may not generalize to other programming programming languages or even to varied patch datasets. The study of Csuvik *et al.* [8] is also closely related to ours since it explores BERT embeddings to define similarity thresholds. Their work however remains preliminary (it does not investigate the discriminative power of features) and has been performed at a very small scale (single pre-trained model on 40 one-line bugs from simple programs).

2.2 Distributed Representation Learning

Learning distributed representations have been widely used to advance several machine learning tasks. In particular, in the field of Natural Language Processing embedding techniques such as Word2Vec [20], **Doc2Vec** [20] and **BERT** [9] have been successfully applied for different semantics-related tasks. By building on the hypothesis of code naturalness [2, 12], a number of software engineering research works have also leveraged the aforementioned approaches for learning distributed representations of code. Alon *et al.* [3] have then proposed **code2vec**, an embedding technique that explores AST paths to take into account structural information in code. More recently, Hoang *et al.* [13] have proposed **CC2Vec**, which further specializes to code changes.

Our work explores different techniques across the spectrum of distributed representation learning. We therefore consider four variants from the seemingly-least specialized to code (i.e., Doc2Vec) to the state of the art for code change representation (i.e., CC2Vec).

2.2.1 **Doc2Vec.** Doc2Vec [20] is an unsupervised framework mostly used to learn continuous distributed vector representations of sentences, paragraphs and documents, regardless of their lengths. It works on the intuition, inspired by the method of learning word vectors [36], that the document representation should be good enough to predict the words in the document Doc2Vec has been applied in various software engineering tasks. For example, Wei and Li [47] leveraged Doc2Vec to exploit deep lexical and syntactical features for software functional clone detection. Ndichu *et al.* [39] employed Doc2Vec to learn code structure representation at AST level to predict JavaScript-based attacks.

2.2.2 **BERT.** BERT [9] is a language representation model that has been introduced by an AI language team in Google. BERT is devoted to pre-train deep bidirectional representations from unlabelled texts. Then a pre-trained BERT model can be fine-tuned to accomplish various natural language processing tasks such as question answering or language inference. Zhou *et al.* [61] employed a BERT pre-trained model to extract deep semantic features from code name information of programs in order to perform code recommendation. Yu *et al.* [59] even leveraged BERT on binary code to identify similar binaries.

2.2.3 **code2vec.** code2vec [3] is an attention-based neural code embedding model developed to represent code fragments as continuous distributed vectors, by training on AST paths and code tokens. Its embeddings have notably been used to predict the semantic properties of code fragments [3], in order, for instance, to predict method names. In a recent work, however, Kang *et al.* [18] reported an empirical study, which highlighted that the yielded token code2vec embeddings may not generalize to other code-related tasks such as code comment generation, code authorship identification or code clone detection. code2vec remains however the state of the art in code embeddings: Compton *et al.* [7] recently leveraged code2vec to embed Java classes and learn code structures for the task of variable naming obfuscation.

2.2.4 **CC2Vec.** CC2Vec [13] is a specialized hierarchical attention neural network model which learns vector representations of code changes (i.e., patches) guided by the associated commit messages (which is used as a semantic representation of the patch). As the authors demonstrated in their in large empirical evaluation, CC2Vec presents promising performance on commit message generation, bug fixing patch identification, and just-in-time defect prediction.

3 STUDY DESIGN

First, we overview the research questions that we investigate. Then we present the datasets that are leveraged to answer these research questions. Finally, we discuss the actual training of (or use of pre-trained) models for embedding the code changes.

3.1 Research Questions

RQ1: *Do different representation learning models yield comparable distributions of similarity values between buggy code and patched code?*

A widespread hypothesis in program repair is that bug fixing generally induce minimal changes [5, 6, 15, 16, 28–30, 35,

49, 50, 54]. We propose to investigate whether embeddings can be a reliable means for assessing the extent of changes through computation of cosine similarity between vector representations.

RQ2: *To what extent similarity distributions can be generalized for inferring a cutoff value to filter out incorrect patches?* Following up on RQ1, We propose in this research question to experiment ranking patches based on cosine similarity of their vector representations, and rely on naively-defined similarity thresholds to decide on filtering of incorrect patches.

RQ3: *Can we learn to predict patch correctness by training classifiers with code embeddings input?* We investigate whether deep learned features are indeed relevant for building machine learning predictors for patch correctness.

3.2 Datasets

We collect patch datasets by building on previous efforts in the community. An initial dataset of correct patches is collected by using five literature benchmarks, namely Bugs.jar [42], Bears [33], Defects4J [17], QuixBugs [26] and ManySStuBs4J [19]. These are developer patches as committed in open source project repositories.

We also consider patches generated by APR tools integrated into the `RepairThemAll` framework. We use all patch samples released by Durieux *et al.* [10]. This only includes sample patches that make the programs pass all test cases. They are thus plausible. However, no validation information on correctness was given. In this work, we proceed to manually validate the generated patches, among which we identified 900 correct patches. The correctness validation follows the criteria defined by Liu *et al.* [31].

In a recent study on the efficiency of program repair, Liu *et al.* [31] released a labeled dataset of patches generated by 16 APR systems for the Defects4J bugs. We consider this dataset as well as the labeled dataset that was used to evaluate the PATCH-SIM [53] approach.

Overall, Table 1 summarizes the data sets that we used for our experiments. Each experiment in Section 4 has specific requirements on the data (e.g., large patch sets for training models, labeled datasets for benchmarking classifiers, etc.). For each experiment, we will recall which sub-dataset has been leveraged and why.

Table 1: Datasets of Java patches used in our experiments.

Subjects	contains incorrect patches	contains correct patches	labelled dataset	# Patches
Bears [33]	No	Yes	-	251
Bugs.jar [42]	No	Yes	-	1,158
Defects4J [17][†]	No	Yes	-	864
ManySStubBs4J [19]	No	Yes	-	34,051
QuixBugs [26]	No	Yes	-	40
RepairThemAll [10]	Yes	Yes	No[‡]	64,293
Liu *et al.* [31]	Yes	Yes	Yes	1,245
Xiong *et al.* [53]	Yes	Yes	Yes	139
Total				102,041

[†]The latest version 2.0.0 of Defects4J is considered in this study.
[‡]The patches are not labeled in [10]. We support the labeling effort in this study by comparing the generated patches against the developer patches. The 2,918 patches for IntroclassJava in [10] are also excluded from our study since IntroClassJava is a lab-built Java benchmark transformed from the C program bugs in small student-written programming assignments from IntroClass [23].

3.3 Model input pre-processing

Samples in our datasets are patches such as the one presented in Figure 1 extracted from the Defects4J dataset. Our investigations with representation learning however require input data about the buggy and patched code. A straightforward approach to derive those inputs would be to consider the code files before and after the patch. Unfortunately, depending on the size of the code file, the differences could be too minimal to be captured by any similarity measurement. To that end, we propose to focus on the code fragment that appears in the patch. Thus, to represent the buggy code fragment (cf. Figure 2), we keep all removed lines (i.e., starting with '-') as well as the patch context lines (i.e., those not starting with either '-', '+' or '@'). Similarly, the patched code fragment (cf. Figure 3) is represented by added lines (i.e., starting with '+') as well as the same context lines. Since tool support for the representation learning techniques BERT, Doc2Vec, and CC2Vec require each input sample to be on a single line, we flatten multi-line code fragments into a single line.

```
--- a/source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.
    java
+++ b/source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java
@@ -1794,7 +1794,7 @@ public abstract class AbstractCategoryItemRenderer
    extends AbstractRenderer
 }
 int index = this.plot.getIndexOf(this);
 CategoryDataset dataset = this.plot.getDataset(index);
-  if (dataset != null) {
+  if (dataset == null) {
     return result;
   }
   int seriesCount = dataset.getRowCount();
```

Figure 1: Example of a patch for the Defects4J bug Chart-1.

```
1: a/source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java
2:     }
3:     int index = this.plot.getIndexOf(this);
4:     CategoryDataset dataset = this.plot.getDataset(index);
5:     if (dataset != null) {
6:       return result;
7:     }
8:     int seriesCount = dataset.getRowCount();
```

Figure 2: Buggy code fragment associated to patch in Fig. 1.

```
1: b/source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java
2:     }
3:     int index = this.plot.getIndexOf(this);
4:     CategoryDataset dataset = this.plot.getDataset(index);
5:     if (dataset == null) {
6:       return result;
7:     }
8:     int seriesCount = dataset.getRowCount();
```

Figure 3: Patched code fragment associated to patch in Fig. 1.

In contrast to BERT, Doc2Vec, and CC2Vec, which can take as input some syntax-incomplete code fragments, code2vec requires the fragment to be fully parsable in order to extract information on Abstract Syntax Tree paths. Since patch datasets include only text-based diffs, code context is generally truncated and is likely not parsable. However, as just explained, we opt to consider only the removed/added lines to build the buggy and patched code input data. By doing so, we substantially improved the success rate of the JavaExtractor tool used to build the tokens in the code2vec pipeline.

3.4 Embedding models

When representation learning algorithms are applied to some training data, they produce *embedding models* that have learned to map a set of code tokens in the vocabulary of the training data to vectors of numerical values. These vectors are also referred to as *embeddings*. Figure 4 illustrates the process of embedding buggy code and patched code for the purpose of our experiments.

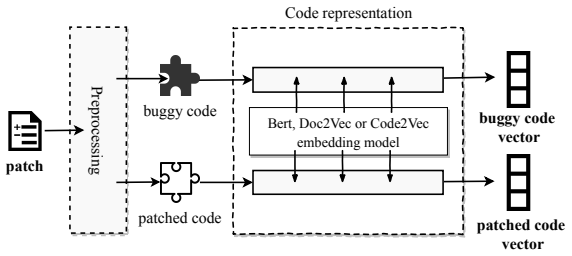

Figure 4: Producing code fragment embeddings with BERT, Doc2Vec and code2vec.

The embedding models used in this work are obtained from different sources and training scenarios.

- **BERT.** In the first scenario, we consider an embedding model that initially targets natural language data, both in terms of the learning algorithm and in terms of training data. The network structure of BERT, however, is deep, meaning that it requires large datasets for training the embedding model. As it is now custom in the literature, we instead leverage a pre-trained 24-layer BERT model, which was trained on a Wikipedia corpus.
- **Doc2Vec.** In the second scenario, we consider an embedding model that is trained on code data but using a representation learning technique that was developed for text data. To that end, we have trained the Doc2Vec model with code data of 36,364 patches from the 5 repair benchmarks (cf. Table 1).
- **code2vec.** In the third scenario, we consider an embedding model that primarily targets code, both in terms of the learning algorithm and in terms of training data. We use in this case a pre-trained model of code2vec, which was trained by the authors using ~14 million code examples from Java projects.
- **CC2Vec.** Finally, in the fourth scenario, we consider an embedding model that was built in representation learning experiments for code changes. However, the pre-trained model that we leveraged from the work of Hoang *et al.* [13] is embedding each patch into a single vector. We investigate the layers and identify the middle CNN-3D layer as the sweet spot to extract embeddings for buggy code and patched code fragments. We illustrated the process in Figure 5.

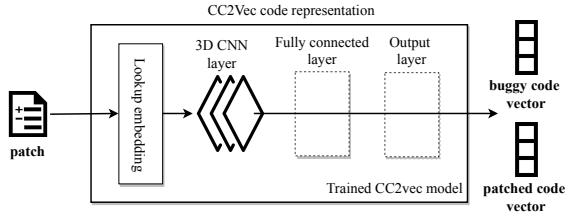

Figure 5: Extracting code fragment embeddings from CC2Vec pre-trained model.

4 EXPERIMENTS

We present the experiments that we designed to answer the research questions of our study. For each experiment, we state the objective, overview the execution details before presenting the results.

4.1 [Similarity Measurements for Buggy and Patched Code using Embeddings]

Objective: We investigate the capability of different learned embeddings to capture the similarity/dissimilarity between code fragments. The experiments are performed towards providing answers for two sub-questions:

RQ-1.1 Is correct code actually similar to buggy code based on learned embeddings?

RQ-1.2 To what extent is buggy code more similar to correctly-patched code than to incorrectly-patched code?

Experimental Design: We perform two distinct experiments with available datasets to answer RQ-1.1 and RQ-1.2.

[Experiment ❶] Using the four embedding models considered in our study (cf. Section 3.4), we produce the embeddings for buggy and patched code fragments associated to 36k patches available in five repair benchmarks (as shown in Table 2). In this case, the patched code fragment represents correct code since it comes from labeled benchmark data (generally representing developer fix patches). Given those embeddings (i.e., code representation vectors), we compute the cosine similarity between the vector representing the buggy code fragment and the vector representing the patched code fragment.

Table 2: Patch datasets used for computing similarity scores between buggy code fragments and correct code fragments.

	Bears	Bugs.jar	Defects4J	ManySStuBs4J	QuixBugs	Total
# Patches	251	1,158	864	34,051	40	36,364[1]

[1] Due to parsing failures, code2vec embeddings are available for 21,135 patches.

[Experiment ❷] To compare the similarity scores of correct code fragment vs incorrect code fragment to the buggy code, we consider combining datasets with correct patches and datasets with incorrect patches. Note that, all patches in our experiments are plausible since we are focused on correctness: plausibility is straightforward to decide based on test suites. Correct patches are provided in benchmarks. However, incorrect patches associated to all benchmark bugs are not available. We rely on the dataset released by Liu *et al.* [31]: 674 plausible but incorrect patches generated by 16 repair tools for 184 Defects4J bugs are considered from this dataset. Those 674 incorrect patches are selected within a larger set of incorrect patches by adding the constraint that the incorrect patch should be changing the same code location as the developer-provided patch in the benchmark: such incorrect patch cases may indeed be the most challenging to identify with heuristics. We propose to compare the similarity scores between the incorrect code and buggy code associated to the dataset with the similarity scores between correct code and buggy associated to all benchmarks, all Defects4J benchmark data, or only the subset of Defects4J that corresponds to the 184 patches for which relevant incorrect patches are available.

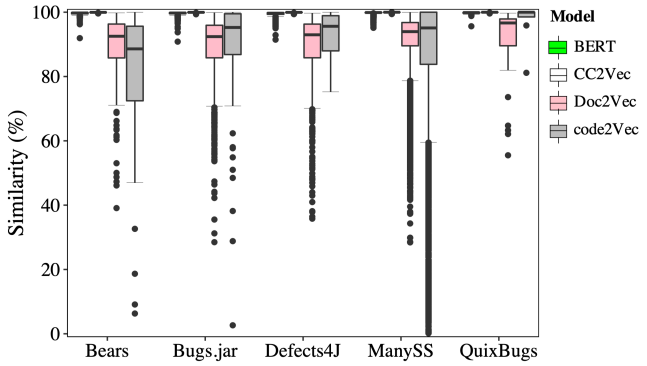

Figure 6: Distributions[1] of similarity scores between correct and buggy code fragments.

[1] "ManySS" stands for "ManySStuBs4J"

Results: Figure 6 presents the boxplots of the similarity distributions with different embedding models and for samples in different datasets. Doc2Vec and code2vec models appear to yield similarity values that are lower than BERT and CC2Vec models.

In Figure 7, we zoom in the boxplot region for each embedding model experiment to overview the differences across different benchmark data. We obverse that, when embedding the patches with BERT, the similarity distribution for the patches in Defects4J dataset is similar to Bugs.jar and Bears dataset, but is different from the dataset ManySStBs4J and QuixBugs. The Mann-Whitney-Wilcoxon (MWW) tests [34, 51] confirm that the similarity of median scores for Defects4J, Bugs.jar and Bears is indeed statistically significant. MWW tests further confirms the statistical significance of the difference between Defects4J and ManySStBs4J/QuixBugs scores.

Defects4J, Bugs.jar and Bears include diverse human-written patches for a large spectrum of bugs from real-world open-source Java projects. In contrast, ManySStuBs4J only contains patches for single statement bugs. Quixbugs dataset is further limited by its size and the fact that the patches are built by simply mutating the code of small Java implementation of 40 algorithms (e.g., quicksort, levenshtein, etc.).

While CC2Vec and Doc2Vec exhibit roughly similar patterns with BERT (although at different scales), the experimental results with code2vec present different patterns across datasets. Note that, due to parsing failures of code2vec, we eventually considered only 118 Bears patches, 123 Bugs.jar patches, 46 Defects4J patches, 20,840 ManySStuBs4J patches and 8 QuixBugs. The change of dataset size could explain the difference with the other embedding models.

✍ **RQ1.1** ▶ *Learned representations of buggy and correct code fragments exhibit high cosine similarity scores. Median scores are similar for patches that are collected with similar heuristics (e.g., in the wild patches vs single-line patches vs debugging example patches). The pre-trained BERT natural language model captures more similarity variations than the CC2Vec model, which is specialized for code changes.*◀

In the second experiment, we further assess whether incorrectly-patched code exhibits different similarity score distributions than correctly-patched code. Figure 8 shows the distributions of cosine

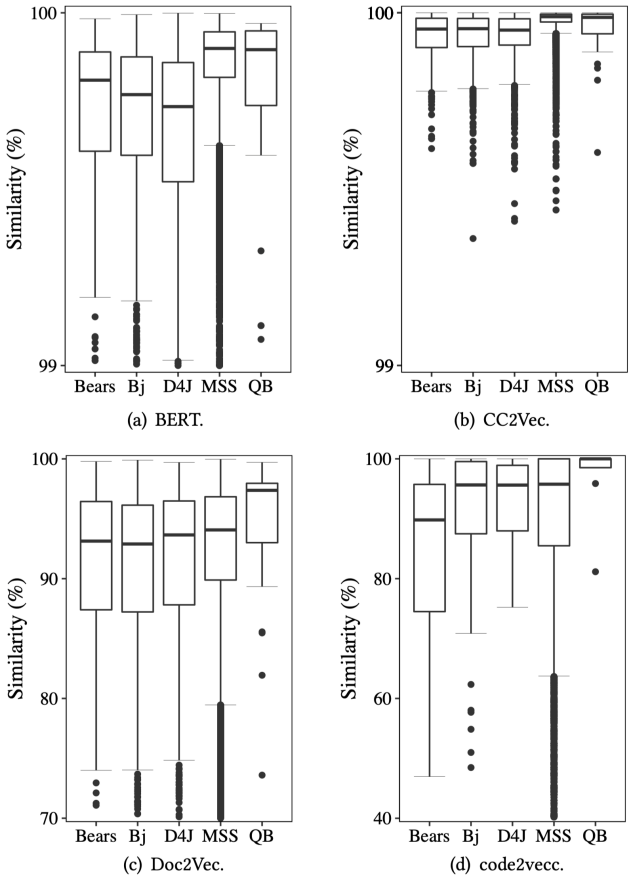

Figure 7: Zoomed views of the distributions of similarity scores between correct and buggy code fragments.

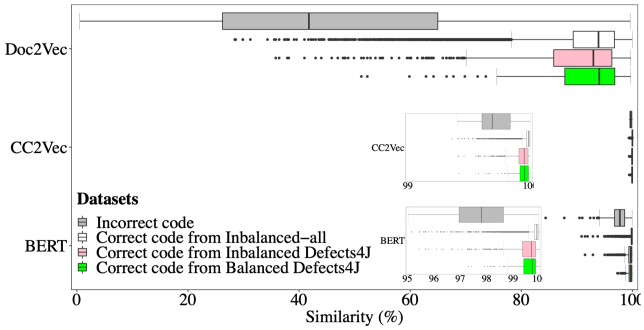

Figure 8: Comparison of similarity score distributions for code fragments in incorrect and correct patches.

similarity scores for correct patches (i.e., similarity between buggy code and correct code fragments) and incorrect patches (i.e., similarity between buggy code and incorrect code fragments). The comparison is done with different scenarios specified in Table 3.

The comparisons do not include the case of embeddings for code2vec. Indeed, unlike the previous experiment where code2vec was able to parse enough code fragments, for the considered 184 correct patches of Defects4J, code2vec failed to parse most of the relevant code fragments. Hence, we focus the comparison on the other three embedding models (pre-trained BERT, trained Doc2Vec and pre-trained CC2Vec). Overall, we observe that the distribution

Table 3: Scenarios for similarity distributions comparison

Scenario	Incorrect patches	Correct patches
Imbalanced-all[1]	674 incorrect patches	all correct patches from 5 benchmarks in Table 2.
Imbalanced-Defects4J	by 16 APR tools [31]	all correct patches from Defects4J.
Balanced-Defects4J	for 184 Defects4J bugs	all correct patches for the 184 Defects4J bugs.

[1] Except for Defects4J, there are no publicly-released incorrect patches for APR benchmarks.

of cosine similarity scores is substantially different for correct and incorrect code.

We observe that the similarity distributions of buggy code and patched code from incorrect patches are significantly different from the similarities for correct patches. The difference of median values is confirmed to be statistically significant by an MWW test. Note that the difference remains high for BERT, Doc2Vec and CC2Vec whether the correct code is the counterpart of the incorrect ones (i.e., the scenario of Balanced-Defects4J) or whether the correct code is from a larger dataset (i.e., Imbalanced-all and Imbalanced-Defects4J scenarios).

✍ **RQ1.2** ▶ *Learned representations of code fragments with BERT, CC2Vec and Doc2Vec yield similarity scores that, given a buggy code, substantially differ between correct code and incorrect code. This result suggests that similarity score can be leveraged to discriminate correct patches from incorrect patches.*◀

4.2 [Filtering of Incorrect Patches based on Similarity Thresholds]

Objective: Following up on the findings related to the first research question, we investigate the selection of cut-off similarity scores to decide on which APR-generated patches are likely incorrect. Results from this investigation will provide insights to guide the exploitation of code embeddings in program repair pipelines.

Experimental design. To select threshold values, we consider the distributions of similarity scores from the above experiments (cf. Section 4.1). Table 4 summarizes relevant statistics on the distributions on the similarity scores distribution for correct patches. Given the differences that were exhibited with incorrect patches in previous experiments, we use, for example, the 1^{st} quartile value as an inferred threshold value.

Table 4: Statistics on the distributions of similarity scores for correct patches of Bears+Bugs.jar+Defects4J

Subjects	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean
BERT	90.84	99.47	99.73	99.86	100	99.54
CC2Vec	99.36	99.91	99.95	99.98	100	99.93
Doc2Vec	28.49	85.80	92.60	96.10	99.89	89.19
code2vec	2.64	81.19	93.63	98.87	100	87.11

Given our previous findings that different datasets exhibit different similarity score distributions, we also consider inferring a specific threshold for the QuixBugs dataset (cf. statistics in Table 5). We do not compute any threshold based on ManySStuBs4J since it has not yet been applied to program repair tools.

Our test data is constituted of 64,293 patches generated by 11 APR tools in the empirical study of Durieux *et al.* [10]. First, we use the four embedding models to generate embeddings of buggy code and patched code fragments and compute cosine similarity scores. Second, for each bug, we rank all generated patches based on the similarity score between the patched code and the buggy, where we consider that the higher the score, the more likely the correctness. Finally, to filter incorrect candidates, we consider two experiments:

Table 5: Statistics on the distributions of similarity scores for correct patches of QuixBugs.

Subjects	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean
BERT	95.63	99.69	99.89	99.95	99.97	99.66
CC2Vec	99.60	99.94	99.99	100	100	99.95
Doc2Vec	55.51	89.56	96.65	97.90	99.72	91.29
code2vec	81.16	98. 53	100	100	100	97.06

(1) Patches that lead to similarity scores that are lower to the inferred threshold (i.e., 1^{st} Quartile in previous experimental data) will be considered as incorrect. Patches where patched code exhibit higher similarity scores than the threshold are considered likely correct.
(2) Another approach is to consider only the top-1 patches with the highest similarity scores as correct patches. Other patches are considered incorrect.

In all cases, we systematically validate the correctness of all 64,293 patches to have the correctness labels, for which the dataset authors did not provide (all plausible patches having been considered as valid). First, if the file(s) modified by a patch are not the same buggy files in the benchmark, we systematically consider it as incorrect: with this simple scheme, 33 489 patches are found incorrect. Second, with the same file, if the patch is not making changes at the same code locations, we consider it to be incorrect: 26 386 patches are further tagged as incorrect with this decision (cf. Threats to validity in Section 5). Finally, for the remaining 4 418 plausible patches in the dataset, we manually validate correctness by following the strict criteria enumerated by Liu *et al.* [31] to enable reproducibility. Overall, we could label 900 correct patches. The remainders are considered as incorrect.

Results. By considering the patch with the highest (top-1) similarity score between the patched code and buggy code as correct, we were able to identify a correct patch for 10% (with BERT), 9% (with CC2Vec) and 10% (with Doc2Vec) of the bug cases. Overall we also misclassified 96% correct patches as incorrect. However, only 1.5% of incorrect patches were misclassified as correct patches.

Given that a given bug can be fixed with several correct patches, the top-1 criterion may not be adequate. Furthermore, this criterion makes the assumption that a correct patch indeed exists among the patch candidates. By using filtering thresholds inferred from previous experiments (which do not include the test dataset in this experiment), we can attempt to filter all incorrect patches generated by APR tools. Filtering results presented in Table 6 show the recall scores that can be reached. We provide experimental results when we use 1^{st} Quartile and Mean values of similarity scores in the "training" set as threshold values. The threshold are also applied by taking into account the datasets: thresholds learned on QuixBugs benchmark are applied to generated patches for QuixBugs bugs.

Table 6: Filtering incorrect patches by generalizing thresholds inferred from Section 4.1.Results.

Dataset	# CP	# IP	Threshold	BERT				CC2Vec				Doc2Vec			
				# +CP	# -IP	+Recall	-Recall	# +CP	# -IP	+Recall	-Recall	# +CP	# -IP	+Recall	-Recall
Bears, Bugs.jar and Defects4J	893	61,932	1st Qu.	57	48,846	6.4%	78.9%	797	19,499	89.2%	31.5%	794	25,192	88.9%	40.7%
			Mean	49	51,783	5.5%	83.6%	789	23,738	88.4%	38.3%	771	33,218	86.3%	53.6%
QuixBugs	7	1,461	1st Qu.	4	1,387	57.1%	94.9%	4	1,198	57.1%	82.0%	7	1,226	100%	83.9%
			Mean	4	1,378	57.1%	94.3%	4	1,255	57.1%	85.9%	7	1270	100%	86.9%

*"**# CP**" and "**# IP**" stand for the number of correct and incorrect patches, respectively. "**# +CP**" means the number of correct patches that can be ranked upon the threshold, while "**# -IP**" means the number of incorrect patches that can be filtered out by the threshold. "**+Recall**" and "**-Recall**" represent the recall of identifying correct patches and filtering out incorrect patches, respectively.

✍ **RQ2** ▶*Building on cosine similarity scores, code fragment embeddings can help to filter out between 31.5% (with CC2Vec) and 94.9% (with BERT) of incorrect patches. While it can achieve the highest recall of filtering incorrect patches, BERT produces embeddings that lead to a lower recall (at 5.5%) at identifying correct patches.*◀

4.3 [Classification of Correct Patches with supervised learning]

Objective. Cosine similarity between embeddings (which was used in the previous experiments) considers every deep learned feature as having the same weight as the others in the embedding vector. We investigate the feasibility to infer, using machine learning, the weights that different features may present with respect to patch correctness. We compare the prediction evaluation results with the achievements of related approaches in the literature.

Experimental design. To perform our machine learning experiments, we first require a ground-truth dataset. To that end, we rely on labeled datasets in the literature. Since incorrect patches generated by actual APR tools are only available for the Defects4J bugs, we focus on labeled patches provided by two independent teams (Liu *et al.* [31] and Xiong *et al.* [53]). Very few patches generated by the different tools are actually labeled as correct, leading to an imbalanced dataset. To reduce the imbalance issue, we supplement the dataset with developer (correct) patches as supplied in the Defects4J benchmark. Eventually, our dataset included 1134 patches. We removed duplicates to avoid data leak bias.

Table 7: Dataset for evaluating ML-based predictors of patch correctness

	Correct patches	Incorrect patches	Total
Liu *et al.* [31]	137	502	639
Xiong *et al.* [53]	30	109	139
Defects4J (developers) [17]	356	0	356
Whole dataset	523	611	1134
Final Dataset (deduplicated)	**468**	**532**	**1000**

Our ground truth dataset patches are then fed to our embedding models to produce embedding vectors. As for previous experiments, the parsability of Defects4J patch code fragments prevented the application of code2vec: we use pre-trained models of BERT (trained with natural language text) and CC2Vec (trained with code changes) as well as a retrained model of Doc2Vec (trained with patches).

Since the representation learning models are applied to code fragments inferred from patches (and not to the patch themselves), we collect the embeddings of both buggy code fragment and patched code fragment for each patch. Then we must merge these vectors back into a single input vector for the classification algorithm. We follow an approach that was demonstrated by Hoang et al. [13] in a recent work on bug fix patch prediction: the classification model performs best when features of patched code fragment and buggy code fragment are crossed together. We thus propose a classification pipeline (cf. Figure 9) where the feature extraction for a given patch is done by applying subtraction, multiplication, cosine similarity and euclidean similarity to capture crossed features between the buggy code vector and the patched code vector. The resulting patch embedding has 2*n+2 dimensions where n is the dimension of input code fragment embeddings.

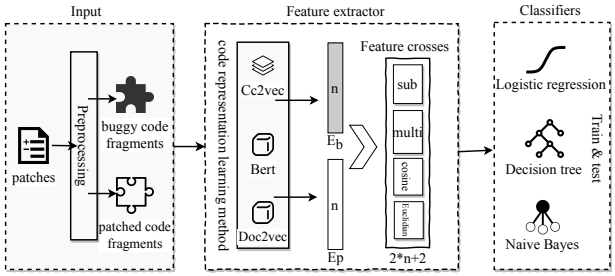

Figure 9: Feature engineering for correctness classification.

Results. We compare the performance of different predictors (varying the embeding models) using different learners (i.e., classification algorithms). Results presented in Table 8 are averaged from a 5-fold cross validation setup. All classical metrics used for assessing predictors are reproted: Accuracy, Precision, Recall, F1-Measure, Area Under Curve (AUC). Logistic Regression (LR) applied to BERT embeddings yield the best performance measurements: 0.720 for F1 and 0.808 for AUC.

Table 8: Evaluation of Bert representation on three ML classifiers.

Classifier	Embedding	Acc.	Prec.	Recall.	F1	AUC
DecisionTree	BERT	63.6	62.0	57.3	59.6	0.632
	CC2Vec	69.0	66.9	68.0	67.2	0.690
	Doc2Vec	60.2	57.4	57.7	57.5	0.600
Logistic regression	BERT	74.4	73.8	70.3	72.0	0.808
	CC2Vec	73.9	72.5	72.0	72.0	0.788
	Doc2Vec	66.3	65.3	59.9	62.3	0.707
Naive bayes	BERT	60.3	55.6	77.0	64.5	0.642
	CC2Vec	58.0	65.4	22.7	28.5	0.722
	Doc2Vec	66.3	69.4	49.8	57.9	0.714

✍ **RQ3.1** ▶ *An ML classifier trained using Logistic Regression with BERT embeddings yield very promising performance on patch correctness prediction (F-Measure at 72.0% and AUC at 80.8%).* ◀

[Comparison against the state of the art]. There are two related works for patch prediction which were both evaluated on 139 patches released by Xiong *et al.* [53]. PATCH-SIM [53] compares execution traces of patched programs to identify correctness. ODS [57] leverages manually-crafted features to build machine learning classifiers.

We consider the 139 patches as test set and the remainder in our dataset (870 = 1000 − 130[1]) for training. Note that the 139 patches are associated to bug cases where repair tools can generate patches. These patches may thus be substantially different from the rest in our dataset. Indeed our best learner (Logistic Regression with BERT embeddings) yields an AUC of 0.765. The Receiver Operating Characteristic (ROC) curve is presented in Figure 10.

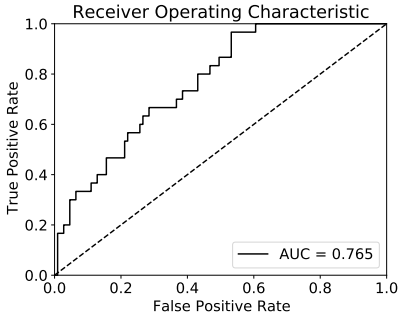

Figure 10: Performance of ML patch correctness predictor using BERT/Logistic Regression: Test set from [53].

In the validation of PATCH-SIM [53], the authors aimed for avoiding to filter out any correct patches. Eventually, when guaranteeing that no correct patch is excluded, they could still exclude 62 (56.3%) incorrect patches. If we constrain the threshold of our predictor to avoid misclassifying any correct patch (threshold value = 0.219), our predictor is able to exclude up to 43 (39.4%) incorrect patches, which represents a reasonably promising achievement since no dynamic information is used (in contrast to PATCH-SIM). Table 9 overviews the prediction results comparison.

Table 9: Comparison of incorrect patch identification between PATCH-SIM (uses dynamic information) and BERT+LR (uses embeddings statically inferred from patches)

	Ground Truth		PATCH-SIM		BERT + LR	
Project	**Incorrect**	**Correct**	**Incorrect excluded (%)**	**Correct excluded**	**Incorrect excluded (%)**	**Correct excluded**
Chart	23	3	14(60.9%)	0	16(69.6%)	0
Lang	10	5	6(54.5%)	0	1(10%)	0
Math	63	20	33(52.4%)	0	23(36.5%)	0
Time	13	2	9(69.2%)	0	3(23.1%)	0
Total	109	30	62(56.3%)	0	43(39.4%)	0

We also compare the predictive power of our models against that of ODS [57], which builds on manually engineered features. We directly compare against the results reported by the authors on the 139 test patches. While the pre-trained BERT model associated with Logistic Regression (LR) achieves better AUC than ODS LR-based model (0.765 vs 0.705), ODS Random Forest-based model achieves a higher AUC at 0.841. Note however that ODS has been trained on over 13 thousand patches (including patches for bugs associated to the test set patch), our training dataset includes only 870 patches (i.e., ~$1/20^{th}$ of their dataset).

Tables 10 and 11 provide confusion matrices for different cut-off thresholds of the classifiers for ODS and our BERT embeddings-based classifiers: TP (true positives) represent correct patches that were classified as such; TN (true negatives) represent incorrect patches that were classified as such; FP (false positives) represent incorrect patches that were classified as correct; and FN (false negatives) represent correct patches that were classified as incorrect. Overall, the BERT-based predictor is very sensitive to the cut-off thresholds while ODS is less sensitive. We also note that BERT embeddings applied to Random Forrest does not yield good performance: decision trees are indeed known to be good for categorical data and request large datasets for training. In our case, the data set is small, while ODS has a training dataset that is about 20 times larger. The hand-crafted features of ODS may also help split the patches into categories while our deep learned features are based on a large vocabulary of natural language text.

Table 10: Confusion matrix of ML predictions based on BERT embedddings with different thresholds.

Learners	**AUC**		**Thresholds**								
			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
LR	0.765	#TP	30	30	24	19	16	12	10	6	4
		#TN	13	37	61	79	85	95	100	106	108
		#FP	96	72	48	30	24	14	9	3	1
		#FN	0	0	6	11	14	18	20	24	26
RF	0.751	#TP	30	30	29	26	20	12	4	2	0
		#TN	1	1	6	32	79	102	107	108	109
		#FP	108	108	103	77	30	7	2	1	0
		#FN	0	0	1	4	10	18	26	28	30

Table 11: Confusion matrix of ODS predictions with different thresholds.

Learners	**AUC**		**Thresholds**								
			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
LR	0.705	#TP	27	27	27	27	27	27	27	27	27
		#TN	50	50	50	50	50	51	51	52	52
		#FP	60	60	60	60	60	59	59	58	58
		#FN	2	2	2	2	2	2	2	2	2
RF	0.841	#TP	29	29	29	29	29	29	25	23	14
		#TN	20	33	36	43	51	60	68	81	101
		#FP	90	77	74	67	59	50	42	29	13
		#FN	0	0	0	0	0	0	4	6	15

We observe nevertheless that LR classifiers fed with BERT embeddings are able to recall high numbers of incorrect patches (#TN is high and #FP is low on threshold > 0.5). In contrast ODS consistently recalls correct patches (however with high false positives). These experimental results suggest that both approaches can be used in a complementary way. In future work, we will propose an approach that carefully merges deep learned features to hand-crafted features towards yielded a better predictors of patch correctness.

✍ **RQ3.2** ►*ML predictors trained on learned representations appear to perform slightly less well than state of the art PATCH-SIM approach which relies on dynamic information. On the other hand, deep code representations appear to be complementary to hand-crafted features engineered for ODS. Overall, we recall that our experimental evaluations are performed in a zero-shot scenario, i.e., without fine-tuning the parameters of any of the pre-trained models. Furthermore, the training dataset of the classifiers is an order of magnitude smaller[a] than the one used by most closely-related work (i.e., ODS) and may further not be representative to best fit the test set.*◄

[a]We were not able to collect or reconstitute the training dataset used in ODS to train our model.

[1]9 patches in the ground truth dataset by Xiong et al. [53] were duplicates (e.g., Patch151 ≡ Patch23).

5 DISCUSSIONS

We enumerate a few insights from our experiments with representation learning models and discuss some threats to validity.

5.1 Experimental insights

[*Code-oriented embedding models may not yield the best embeddings for training predictors.*] Our experiments have revealed that the BERT model which was pre-trained on Wikipedia is yielding the best recall in the identification of incorrect patches. There are several possible reasons to that: Bert implements the deepest neural network and builds on the largest training data. Its performance suggests that code-oriented embeddings should aim for being accurate with small training datasets in order to become competitive against BERT. While we were completing the experiments, a pre-trained CodeBERT [11] model has been released (on April 27). In future work, we will investigate its relevance for producing embeddings that may yield higher performance in patch correctness prediction. In any case, we note that CC2Vec provided the best embeddings for yielding the best recall in identifying correct patches (using similarity thresholds). This suggests that future research should investigate the value of merging different representations or combining the eventual prediction probabilities to improve performance in identifying both correct patches while excluding most incorrect patches.

[*The small sizes of the code fragments lead to similar embeddings.*]. Figure 11 illustrates the different cosine similarity scores that can be obtained for the BERT embeddings of different pairs of short sentences. Although the sentences are semantically (dis)similar, the cosine similarity scores are quite close. This explains why recalling correct patches based on a similarity threshold was a failed attempt (~ 5% for APR-generated patches for. Defects4J+Bears+Bugs.jar bugs). Nevertheless, experimental results demonstrated that deep learned features were relevant for learning to discriminate.

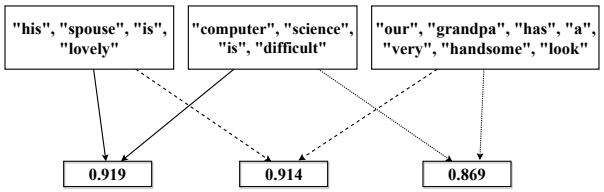

Figure 11: Close cosine similarity scores with small-sized inputs for BERT embedding model.

[*Embeddings are most suitable when applied to simple ML algorithms.*] Because embeddings are yielded from neural networks, they are actually formed by complex crossed features. When they are fed to a complex discriminant model such as Random Forrest, it may lead to overfitting with small datasets. Our experiments however show that simple Logistic Regression yields the best AUC, suggesting that this learner was able to better identifying discriminating features for the prediction task.

5.2 Threats to validity

Our empirical study carries a number of threats to validity that we have tried to mitigate.

Threats to External validity. There are a variety of representation learning models in the literature. A threat to validity of our study is that we may have a selection bias by considering only four embedding models. We have mitigated this threat by considering representative models in different scenarios (pre-trained vs retrained, code change specific vs code-specific vs natural language oriented).

Another threat to validity is related to the use of Defects4J data in evaluating the ML classifiers. This choice however was dictated by the data available and the aim to compare against related work.

Finally, with respect to the explored models, the attention system of CC2Vec requires some execution parameters to perform well. Since the relevant code was not available, we use use a non-attention version instead, potentially making CC2Vec embeddings be under-performing. We release the artifacts for future comparisons by the research community. *Threats to Internal validity.*

A major threat to internal validity lies in the manual assessment heuristics that we applied to the RepairThemAll-generated dataset. We may have misclassified some patches due to mistakes or conservatism. This threat however holds for all APR work that relies on manual assessment. We mitigate this threat by following clear and reproducible decision criteria, and by further releasing our labelled datasets fro the community to review[2].

Threats to Construct validity. For our experiment, the considered embedding models are not perfect and they may have been under-trained for the prediction task that we envisioned. For this reason, the results that we have reported are likely an underestimation of the capability of representation learning models to capture discriminative features for the prediction of patch correctness. Our future studies on representation learning will address this threat by considering different re-training experiments.

6 RELATED WORK

Analyzing Patch Correctness: To assess the performance of fixing bugs of repair tools and approaches, checking the correctness of patches is key, but not trivial. However, this task was largely ignored or unconcerned in the community until the analysis study of patch correctness conducted by Qi *et al.* [41]. Thanks to their systematic analysis of the patches reported by three generate-and-validate program repair systems (i.e., GenProg, RSRepair and AE), they shown that the overwhelming majority of the generated patches are not correct but just overfit the test inputs in the test suites of buggy programs. In another study, Smith *et al.* [44] uncover that patches generated with lower coverage test suites overfit more. Actually, these overfitting patches often simply break under-tested functionalities, and some of them even make the "patched" program worse than the un-patched program. Since then, the overfitting issue has been widely studied in the literature. For example, Le *et al.* [22] revisit the overfitting problem in semantics-based APR systems. In [21], they further assess the reliability of authors and automated annotations in assessing patch correctness. They recommend to make publicly available to the community the patch correctness evaluations of the authors. Yang and Yang [55] explore the difference between the runtime behavior of programs patched

[2] see: https://anonymous.4open.science/r/cdd9881b-7be2-4afe-afd3-6f8e050d0bbb

with developer's patches and those by APR-generated plausible patches. They unveil that the majority of the APR-generated plausible patches leads to different runtime behaviors compared to correct patches.

Predicting Patch Correctness: To predict the correctness of patches, one of the first explored research directions relied on the idea of augmenting test inputs, i.e., more tests need to be proposed. Yang *et al.* [56] design a framework to detect overfitting patches. This framework leverages fuzz strategies on existing test cases in order to automatically generate new test inputs. In addition, it leverages additional oracles (i.e., memory-safety oracles) to improve the validation of APR-generated patches. In a contemporary study, Xin and Reiss [52] also explored to generate new test inputs, with the syntactic differences between the buggy code and its patched code, for validating the correctness of APR-generated patches. As complemented by Xiong *et al.* [53], they proposed to assess the patch correctness of APR systems by leveraging the automated generation of new test cases and measuring behavior similarity of the failing tests on buggy and patched programs.

Through an empirical investigation, Yu *et al.* [60] summarized two common overfitting issues: incomplete fixing and regression introduction. To assist alleviating the overfitting issue for synthesis-based APR systems, they further proposed `UnsatGuided` that relies on additional generated test cases to strengthen patch synthesis, and thus reduce the generation of incorrect overfitting patches.

Predicting patch correctness with thanks to an augmented set of test cases heavily relies on the quality of tests. In practice, tests with high coverage might be unavailable [57]. In our paper, we do not rely on any new test cases to assess patch correctness, but leverage representation learning techniques to build representation vectors for buggy and patched code of APR-generated patches.

To predict overfitting patches yielded by APR tools, Ye *et al.* [57] propose ODS, an overfitting detection system. ODS first statically extracts 4,199 code features at the AST level from the buggy code and generated patch code of APR-generated patches. Those features are fed into three machine learning algorithms (logistic regression, KNN, and random forest) to learn an ensemble probabilistic model for classifying and ranking potentially overfitting patches. To evaluate the performance of ODS, the authors considered 19,253 training samples and 713 testing samples from the Durieux *et al.* empirical study [10]. With these settings, ODS is capable of detecting 57% of overfitting patches. The ODS approach relates to our study since both leverage machine learning and static features. However, ODS only relies on manually identified features which may not generalize to other programming languages or even other datasets.

In a recent work, Csuvik *et al.* [8] exploit the textual and structural similarity between the buggy code and the APR-patched code with two representation learning models (BERT [9] and Doc2Vec [20]) by considering three patch code representation (i.e., source code, abstract syntax tree and identifiers). Their results show that the source code representation is likely to be more effective in correct patch identification than the other two representations, and the similarity-based patch validation can filter out incorrect patches for APR tools. However, to assess the performance of the approach, only 64 patches from QuixBugs [58] have been considered (including 14 in-the-lab bugs). This low number of considered patches raises questions about the generalization of the approach for fixing bugs in the wild. Moreover, unlike our study, new representation learning models (code2vec [3] and CC2Vec [13]) dedicated to code representation have not been exploited.

Representation Learning for Program Repair Tasks: In the literature, representation learning techniques have been widely explored to boost program repair tasks. Long and Rinard explored the topic of learning correct code for patch generation [32]. Their approach learns code transformation for three kinds of bugs from their related human-written patches. After mining the most recent 100 bug-fixing commits from each of the 500 most popular Java projects, Soto and Le Goues [45] have built a probabilistic model to predict bug fixes for program repair. To identify stable Linux patches, Hoang *et al.* [14] proposed a hierarchical deep learning-based method with features extracted from both commit messages and commit code. Bader *et al.* [4] have proposed Getafix to learn recurring fix patterns from human-written patches and suggest fixes. Our paper is not aiming at proposing a new automated patch generation approach. We indeed rather focus on assessing representation learning techniques for predicting correctness of patches generated by program repair tools.

7 CONCLUSION

In this paper, we investigated the feasibility of statically predicting patch correctness by leveraging representation learning models and supervised learning algorithms. The objective is to provide insights for the APR research community towards improving the quality of repair candidates generated by APR tools. To that end, we, first investigated the use of different distributed representation learning to capture the similarity/dissimilarity between buggy and patched code fragments. These experiments gave similarity scores that substantially differ for across embedding models such as BERT, Doc2Vec, code2vec and CC2Vec. Building on these results and in order to guide the exploitation of code embeddings in program repair pipelines, we investigated in subsequent experiments the selection of cut-off similarity scores to decide which APR-generated patches are likely incorrect. This allowed us to filter out between 31.5% and 94.9% incorrect patches based on brute cosine similarity scores. Finally, we investigated the discriminative power of the deep learned features by training machine learning classifiers to predict correct Patches. DecisionTree, Logistic Regression and Naive Bayes are tried with code embeddings from BERT, Doc2Vec and CC2Vec. Logistic Regression with BERT embeddings yielded very promising performance on patch correctness prediction with metrics like F-Measure at 0.72% and AUC at 0.8% on a labeled deduplicated dataset of 1000 patches. We further showed that the performance of these models on static features is promising when comparing against the state of the art (PATCH-SIM [53]), which uses dynamic execution traces. Experimental results suggests that the deep learned features can be complementary to hand-crafted features (such as those engineered by ODS [57]).

Availability. All artifacts of this study are available in the following anonymous repository:

https://anonymous.4open.science/r/cdd9881b-7be2-4afe-afd3-6f8e050d0bbb

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 281–293. https://doi.org/10.1145/2635868.2635883

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4 (2018), 81:1–81:37. https://doi.org/10.1145/3212695

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[4] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 159:1–159:27. https://doi.org/10.1145/3360585

[5] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 306–317. https://doi.org/10.1145/2635868.2635898

[6] Junjie Chen, Alastair F. Donaldson, Andreas Zeller, and Hongyu Zhang. 2017. Testing and Verification of Compilers (Dagstuhl Seminar 17502). *Dagstuhl Reports* 7, 12 (2017), 50–65. https://doi.org/10.4230/DagRep.7.12.50

[7] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding Java Classes with code2vec: Improvements from Variable Obfuscation. *CoRR* abs/2004.02942 (2020). https://arxiv.org/abs/2004.02942

[8] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *Proceedings of the 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 18–25.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[10] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. *arXiv preprint arXiv:1905.11973* (2019).

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155* (2020).

[12] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*. 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[13] Thong Hoang, Hong Jin Kang, Julia Lawall, and David Lo. 2020. CC2Vec: Distributed Representations of Code Changes. In *Proceedings of the 42nd International Conference on Software Engineering*. IEEE.

[14] Thong Hoang, Julia Lawall, Yuan Tian, Richard Jayadi Oentaryo, and David Lo. 2019. PatchNet: Hierarchical Deep Learning-Based Stable Patch Identification for the Linux Kernel. *CoRR* abs/1911.03576 (2019). http://arxiv.org/abs/1911.03576

[15] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations From Singular Examples via Big Code. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 255–266.

[16] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 298–309.

[17] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.

[18] Hong Jin Kang, Tegawendé F. Bissyandé, and David Lo. 2019. Assessing the Generalizability of Code2vec Token Embeddings. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 1–12. https://doi.org/10.1109/ASE.2019.00011

[19] Rafael-Michael Karampatsis and Charles A. Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *Proceedings of the 17th Mining Software Repositories*. IEEE. http://arxiv.org/abs/1905.13334

[20] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31th International Conference on Machine Learning*. 1188–1196.

[21] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 524–535.

[22] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033.

[23] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[25] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* (2019).

[26] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 55–56.

[27] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*. 1–12. https://doi.org/10.1109/ICSE.2019.00019

[28] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 275–286.

[29] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 456–467.

[30] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42.

[31] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the 42nd International Conference on Software Engineering*. IEEE.

[32] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Vol. 51. ACM, 298–312.

[33] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 468–478.

[34] Henry B Mann and Donald R. Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[35] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205.

[36] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[37] Martin Monperrus. 2009. The living review on automated program repair. In *Proceedings of the Symposium on the Foundations of Software Engineering*. HAL/archives-ouvertes. fr, 315–324.

[38] Martin Monperrus. 2018. Automatic software repair: A bibliography. *Comput. Surveys* 51, 1 (2018), 17:1–17:24.

[39] Samuel Ndichu, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. 2019. A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors. *Applied Soft Computing* 84 (2019). https://doi.org/10.1016/j.asoc.2019.105721

[40] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 254–265.

[41] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 24–36.

[42] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, 10–13.

[43] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 13–24.

[44] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543.

[45] Mauricio Soto and Claire Le Goues. 2018. Using a probabilistic model to predict bug fixes. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 221–231.

[46] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*. 297–308. https://doi.org/10.1145/2884781.2884804

[47] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 3034–3040. https://doi.org/10.24963/ijcai.2017/423

[48] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 356–366.

[49] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 364–374.

[50] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1–11.

[51] F. Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945).

[52] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 226–236.

[53] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 789–799.

[54] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 416–426.

[55] Bo Yang and Jinqiu Yang. 2020. Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level. In *Proceedings of the 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 1–8.

[56] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 831–841.

[57] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. Automated Classification of Overfitting Patches with Statically Extracted Code Features. *CoRR* abs/1910.12057 (2019). http://arxiv.org/abs/1910.12057

[58] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. In *Proceedings of the 1st International Workshop on Intelligent Bug Fixing*. IEEE, 1–10.

[59] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. (2020). https://keenlab.tencent.com/en/whitepapers/Ordermatters.pdf

[60] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering* 24, 1 (2019), 33–67. https://doi.org/10.1007/s10664-018-9619-4

[61] Shufan Zhou, Beijun Shen, and Hao Zhong. 2019. Lancer: Your Code Tell Me What You Need. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 1202–1205. https://doi.org/10.1109/ASE.2019.00137